34 P

NASA Technical Memorandum 109092

# Common Spaceborne Multicomputer Operating System and Development Environment

L. G. Craymer and B. F. Lewis
*Jet Propulsion Laboratory, Pasadena, California*

P. J. Hayes and R. L. Jones
*Langley Research Center, Hampton, Virginia*



February 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

# TABLE OF CONTENTS

# APPENDICES



# LIST OF FIGURES



**NOTE:**

*The use of brand names in this document is for completeness and does not imply NASA endorsement.*

# Acknowledgements

The front cover lists the primary authors of this document, but there are many others involved in the COSMOS effort over the years who have reviewed it, or who have edited it and made substantial revisions and significant improvements to the document.

| | |
|---|---|
| • Asa Andrews, | CTA INCORPORATED |
| • Leon J. Alkalaj, | Jet Propulsion Laboratory |
| • David Cummings, | Jet Propulsion Laboratory |
| • Steve Levoe, | Jet Propulsion Laboratory |
| • Don D. Meyer, | Jet Propulsion Laboratory |
| • Karl M. Schneider, | Jet Propulsion Laboratory |
| • Harry F. Benz, | Langley Research Center |
| • Steve R. Ruggles, | Langley Research Center |
| • Mahyar R. Malekpour, | Lockheed Engineering And Sciences Company |
| • Sukhamoy Som, | Lockheed Engineering And Sciences Company |
| • Rodrigo Obando, | Old Dominion University |
| • Jack Stoughton, | Old Dominion University |
| • Douglas Blough, | University of California at Irvine |
| • Krishna M. Kavi, | University of Texas at Arlington |

The research described in this document was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (NASA), and by the NASA Langley Research Center.

# 1. Introduction

The purpose of this document is to describe the Common Spaceborne Multicomputer Operating System (COSMOS), a software operating system designed for NASA flight missions. This document is preliminary. Reader feedback is welcomed to help guide future refinements.

## 1.1. Objectives

COSMOS has focused on the following issues:

- system fault-tolerance and long-term survivability with graceful degradation;
- dynamic code-patching (i.e. software updating);
- real-time processing of spacecraft subsystem operations;
- multiprocessing issues, such as load balancing over a common pool of processors;
- an integrated programming environment with software tools to reduce the cost and time for the development and maintenance of spacecraft applications.

Fault-Tolerance. COSMOS provides software-based fault-tolerance in four ways. First, software voting may be specified for critical processes. Second, COSMOS maintains a consistent global state on each computer. Therefore, any computer could execute an enabled process. Third, COSMOS confines errors to process boundaries: the global state of the system is not modified except at a point where a process has successfully completed execution. Fourth, COSMOS supports checkpointing with subsequent rollback. This feature is useful for missions with less stringent fault-tolerance requirements.

Dynamic Code Patching. The concept of code and data migration is fundamental to the COSMOS design. This provides inherent support for the incremental replacement of application code.

Historically, the onboard software for unmanned spacecraft has been patched during flight -- at communication rates of as low as 10 bits per second. A reliable method for incremental replacement of application code is a necessity.

Real-Time Processing. COSMOS supports both synchronous and asynchronous scheduling of processes. It is possible to specify fixed, deterministic execution schedules. It is also possible to specify repetition rates and times of initiation and termination. Times may be mission elapsed time, relative to when the specification is commanded, or relative to the start or completion of a designated process. When such timing attributes are not specified, scheduling is asynchronously based on the availability of inputs and resources.

Multiprocessing Environment. The COSMOS computing environment is a distributed set of homogeneous and heterogeneous computers. These computers may be located in spacecraft subsystems with high data-rate instruments. Hardware modules can be added or removed dynamically without system interruption or shut-down. The COSMOS kernel is present on each computer; each computer maintains a copy of the global state and participates in the load sharing. There is no central controller.

Programming Environment. COSMOS provides end-to-end support for programmers with a variety of software tools (see Section 7), including performance ehancement and debugging tools. COSMOS provides an easy to use visual programming interface which gives improved visibility into spacecraft program behavior and also supports fault-tolerance features of COSMOS. See Figure 0 and the next section.

## 1.2. COSMOS Overview

An application program -- such as the Mars rover control program shown in Figure 0 -- is called a "graph." A graph consists of boxes which are interconnected by arcs. Data flows from a box output to an arc to a box input; along the journey the data is packaged into what is called a "token." The presence of a token on an arc is represented by a filled circle on the arc.

The boxes in Figure 0 represent processes. Instantiation of a process is possible when all inputs have tokens. When an instantiation completes, tokens are placed on outputs.

The programmer creates graphs by using a tool called "The Dataflow Editor". The programmer literally draws arrows on the screen to specify the flow of data from one process to another. The programmer uses a conventional language such as C to write source code for the processes. Process source code must follow a few conventions in order to be compatible with COSMOS.

COSMOS graphs may be decomposed into dataflow graphs and control graphs (control graphs also follow a "dataflow" paradigm, but the distinction between graph types is convenient). Dataflow graphs are immutable: only the tokens change over time. Dataflow graphs represent the fundamentals of an application.

Control graphs, on the other hand, represent artificial data dependencies that change as the operating environment changes. Control graphs mold the pattern of execution within a dataflow graph to enhance predictability. They are superimposed over dataflow graphs.

Section 6 provides further details on control graphs. Further details on the COSMOS programming language and environment are available in Sections 5 and 7.

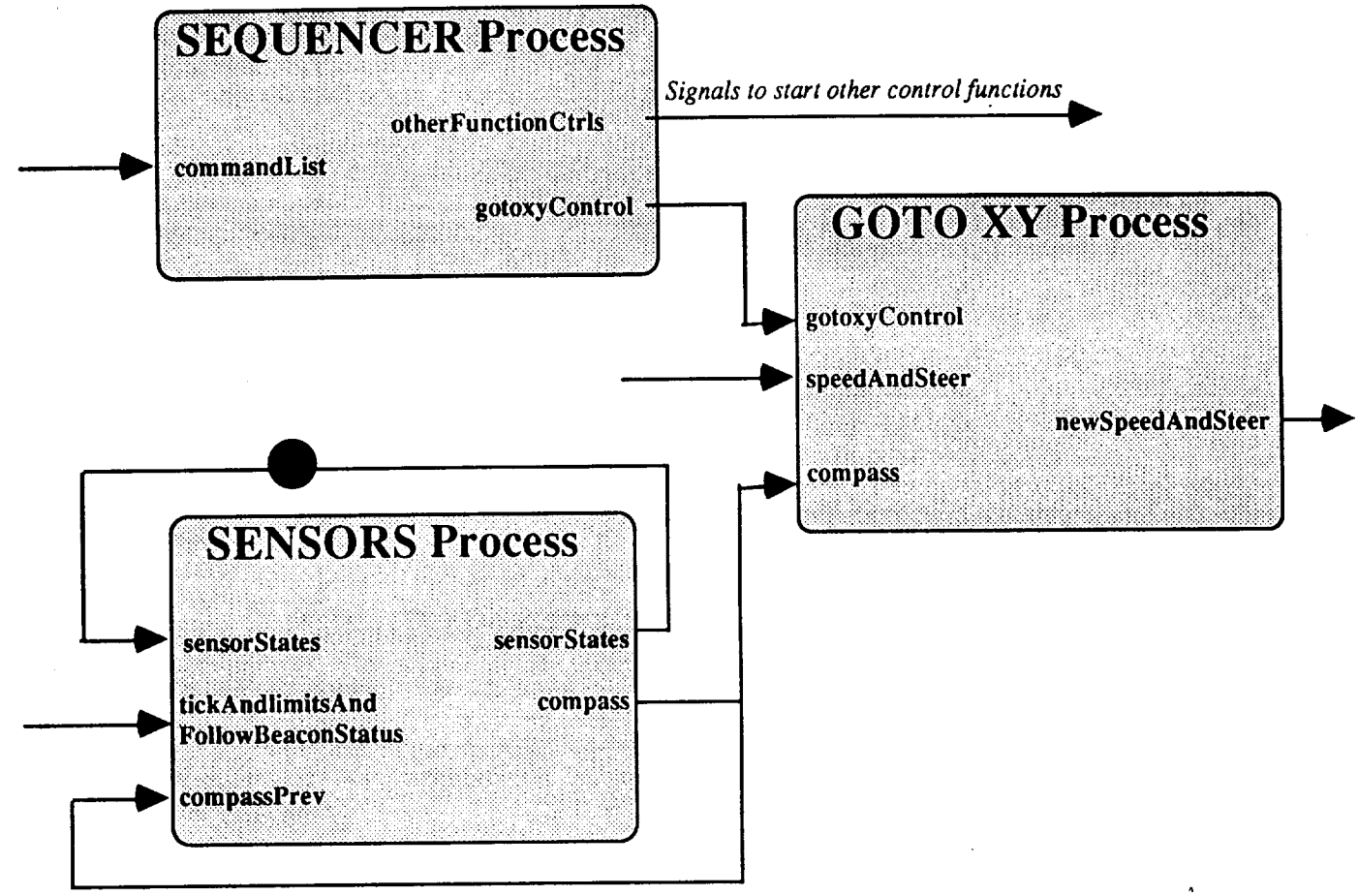


Figure 0. Simplified Excerpt of a Martian Robot Rover Program Written In COSMOS Dataflow.

# 2. Background

Sponsored by NASA's Data Systems Program of the Civil Space Technology Initiative (CSTI), the COSMOS design evolved from multicomputer spacecraft technology developed by the Advanced Flight Computer Group at the Jet Propulsion Laboratory (JPL) and the Multiprocessing Technology Group at NASA's Langley Research Center (LaRC). The operating system developed by JPL (called HYPHOS) and the operating system developed by LaRC (called AMOS) are both based on the dataflow programming paradigm, although the reasons for this are quite different. JPL was attracted to dataflow as a possible solution to problems of spaceborne computing: fault-tolerance (a dataflow system is "always" in a defined state), code-patching, and minimization of hardware redundancy by providing a common pool of processors with transparent multiprocessing rather than the traditional approach of replicated subsystems. LaRC was attracted to dataflow as a basis for predictable, reliable real-time systems for control applications, and has focused on optimizing performance of real-time control applications. The concepts and strategies of each are merged to form COSMOS.

## 2.1. HYPHOS

HYPHOS operating system development has been focused on the general-purpose features needed by independent and cooperating spacecraft subsystems operating asynchronously on an event-driven basis.

The fault-tolerant operating system was developed for the MAX general-purpose fault-tolerant multicomputer, also developed by JPL. The first generation MAX was developed as four interconnected computers based on Motorola 68000 microprocessors. Ten computers are now operational for the second generation MAX system which was designed with space-qualifiability in mind. Each computer uses two National Semiconductor 32032 microprocessors.

The HYPHOS system provides development tools to design dataflow graphs, to diagnose execution traces, and to ensure reliable software implementation.

## 2.2. AMOS

Preceding the design of the operating system, LaRC first proposed an Algorithm To Architecture Mapping Model (ATAMM), which is a Petri net based multicomputer strategy for mapping graph nodes onto computers in a manner which is predictable and deadlock free [1]. It is applicable to both data-driven and to data-independent graphs[1].

The ATAMM Multicomputer Operating System (AMOS) focuses on providing highly predictable and optimal real-time performance of periodically executing large-grain processes in a dataflow architecture, both with and without dynamically changing the number of available computers. For a given number of computers, an operating point defines a schedule of dataflow execution. Such a schedule is enforced by controlling the input data injection rate and by using control arcs between graph nodes as necessary. Control arcs force a specified execution sequence among graph nodes, even when the nodes have no data dependencies among them. Signal processing and control applications are among the candidate applications.

The first generation of AMOS was implemented on the Westinghouse 4-computer 1750A VHSIC Advanced Development Model (ADM) [2]. A second generation version of AMOS has been implemented on a 4-computer GVSC development platform [3,4,5,6]. The GVSC is also based on the 1750A instruction set architecture. Both systems utilize a Parallel Interprocessor (PI) bus.

AMOS provides various software development and diagnostic tools. The ATAMM Environment Tools include a Design Tool for analyzing and modifying graphs, determining the optimum performance bounds versus the number of computers, and determining specific operating points for user-selected performance. The Dataflow Modeller provides simulation of graph execution. The Dataflow Analyzer plays back the results of either simulations or actual hardware executions (including the cases where the number of available computers has changed during execution). The Graph Entry Tool is used to draw and edit graphs.

## 2.3. Merging HYPHOS and AMOS

Neither HYPHOS nor AMOS alone meet all the necessary requirements for embedded spacecraft control, command, and data handling. COSMOS merges the general purpose features of HYPHOS with the real-time predictability and control of AMOS.

COSMOS, when implemented on suitable computing hardware, provides a user-friendly algorithm design, mapping and execution environment for a wide range of spaceborne applications. Development is ongoing to enhance performance predictability for combinations of data-independent and data-dependent dataflow graphs.

---

[1] Data-independent implies decision free Petri-nets (i.e., Marked graphs).

# 3. System Requirements

## 3.1 Development Environment

The COSMOS development environment will run on a UNIX workstation which supports X Windows (Release 4 or later) and the Motif Window Manager (Release 1.1.1 or later).

## 3.2. Target Hardware

COSMOS is designed to function on a wide variety of multicomputer hardware platforms. Only one high-speed intercomputer communication mechanism is strictly necessary. For performance reasons it is desirable to have two separate interconnection networks: one dedicated to the fault tolerant broadcasting of operating system synchronization messages, and the other for data transmission.

Intended to address the needs of a wide variety of potential customers, COSMOS can be implemented -- at some cost in fault coverage -- on a shared memory multicomputer system, a network of UNIX workstations, or even on a single processor. There is probably a performance benefit from a shared memory configuration, and the single processor implementation sacrifices fault-tolerance but gains in relative performance since interprocessor overhead vanishes. However we will assume in this document the presence of a loosely-coupled set of computers which communicate by message passing.

### 3.2.1 Message Passing

The computers may be heterogeneous, but all computers must use a common message passing mechanism. If the system includes special purpose computers which are unsuitable for running a COSMOS kernel -- such as high data rate signal processors -- then it is assumed that these will be controlled by general purpose computers.

The broadcast mechanism used by any particular implementation of COSMOS will depend upon the available communications hardware. To maximize fault-tolerance, the broadcast mechanism should be "Byzantine resilient" (see Section 4.1), and COSMOS assumes that this is the case. However the Byzantine resilient bus may be a physical or a virtual bus, depending on the target hardware system. The operating system design is properly segmented so that the message broadcasting software may be easily changed -- in fact, a version of COSMOS is currently being implemented on a UNIX workstation network.

### 3.2.2 Operating System Kernel

COSMOS can be designed to execute on a bare machine, provided the usual multitasking services listed below are implemented.

- Multiple process threads.
- Priority controlled, event-driven process scheduling.
- Interprocess communication and synchronization, e.g. mailboxes, queues, and semaphores.
- Time management.
- Hooks for dynamic memory management.
- Interrupt handler services.

Priority-driven multitasking is important because it allows rapid response to changing system demands.

Dynamic memory management is used to manage memory for process code. It is also used for data tokens whose memory is not statically assigned.

COSMOS can also be implemented on top of an off-the-shelf kernel that supports these services. For example, the HYPHOS system uses the VRTX kernel from Ready Systems, Inc. This kernel is available for a wide variety of microprocessors, including 1750A instruction set processors.

The AMOS implementations have used the InterACT Ada compiler for the 1750A Westinghouse ADM system and the TLD Ada compiler for the GVSC 1750A system. Both of these systems have used a modified Ada kernel.

### 3.2.3. System Clock

By default COSMOS maintains a global system time without the use of a master clock, but a master/slave clock synchronization procedure can be added to COSMOS at the application level. Reference [7] addresses the tradeoffs between the master/slave and the distributed clock approach.

# 4. A Model Of Implementation

This section describes the current implementation model of dataflow management in COSMOS. The model assumes Byzantine resilient message broadcast, allowing each computer to maintain a consistent global graph state.

## 4.1. Message Passing Model

The two types of messages that are passed among the computers are data messages (tokens) and synchronization messages. Control tokens are included in the synchronization messages.

Data Messages. Data messages are used to transfer data between computers. If a computer needs data from another computer, it opens a logical channel to that computer and sends it a message requesting the data. The message identifies the desired data with a *handle* rather than with a physical address. This allows the receiving computer to check the validity of the request before the data (or an error message) is returned to the requester. The requester then breaks the channel between the computers.

A computer requesting a data block will already know the corresponding checksum value since that was broadcast in the synchronization message that announced the token. The data transfer mechanism does not need to be fault tolerant, since data transfer errors are detected when the message is received; the only restriction imposed is that no computer be able to modify data in another computer and that a path can be found to route messages from one computer to another.

Synchronization Messages. Synchronization messages are used by the operating system to coordinate the activities of the computers. These messages are short, typically about 30-40 bytes. Synchronization messages can be broadcast to all computers or to a designated subset.

COSMOS assumes that the broadcast of synchronization messages is Byzantine resilient [8]. That is, each computer must know whether or not all computers successfully received a message. In addition, messages must be received in the same order by all computers. Each message must identify the broadcasting computer in a way such that neither application software nor hardware errors can cause a message to be mislabeled as coming from some other computer.

In JPL's MAX multicomputer system, all computers are connected to a redundant serial bus with Byzantine resilience, called the Globalbus [9,10].

In LaRC's ADM and GVSC systems, a PI-bus is used to link computers which have a distributed memory. These hardware systems are intended to be within a fault confinement region. A computer performs operations similar to a MAX synchronous broadcast by obtaining exclusive access to the graph, updating the graph state in all computers, and then releasing graph access. The GVSC employs EDAC (error detecting and correcting) memory and provides automatic retry in the case of memory access failure.

A similar but more fault-tolerant approach is being used in the 16-bit Advanced Spaceborne Computer Module (ASCM) and in the 32-bit Advanced Technology Insertion Module (ATIM) currently being space qualified by the Air Force. These are VHSIC hardware candidates for future COSMOS implementations.

## 4.2. Software Model

Processes represent the basic unit of concurrency in COSMOS. Multiple processes may be scheduled for concurrent execution. The graph of inter-process dependencies is referred to as the dataflow graph, with each process represented as a graph node. Dependencies between nodes are represented by arcs connecting the nodes. The dataflow graph also includes the specification of various properties of the processes such as data sizes, memory requirements, etc.

The execution of a dataflow graph is data driven: a process is available for execution only when all necessary inputs to the node are available. These inputs include data tokens and control tokens. (Code for the graph node is packaged in a specialized token, a "code" token.)

Each computer in a COSMOS system maintains a description of the entire graph and its current state. All changes to the graph state (e.g., process acquisition, process completion and announcement of outputs) are made in response to broadcast synchronization messages. All of the computers, including the broadcaster, make the graph state changes when the broadcast is successfully completed. This ensures that a consistent graph state is maintained among all computers[2].

## 4.3. Computer Groups

The computers in a COSMOS system are logically divided into one or more "Computer Groups." These groups may overlap (a computer may belong to more than one group) and group memberships may be changed at runtime by broadcasting synchronization messages.

---

[2] Alternative approaches may be appropriate if shared memory and/or single-bus multicomputer systems are used.

Computer groups are used to control the allocation of processes to computers. Within the dataflow graph, each process is assigned to a single computer group. Only computers within the specified group may execute the process.

If a process is assigned to a computer group containing only one computer, the process is statically assigned to that computer.

For redundant software voting, each instance of a process is assigned to a different computer group. The different computer groups for the process must not overlap.

In heterogeneous systems, different types of computers are assigned to non-overlapping computer groups.

## 4.4. Time Management

By default COSMOS maintains a global system time without the use of a master clock. Each computer regularly broadcasts a synchronization message containing its version of the global time, allowing the computers to converge on a consistent global time. Each computer may also maintain a local time for managing services that are strictly local to a computer such as program delays and I/O time-outs.

For graphs that contain control arcs, it is possible to specify the repetition rate of graphs, as well as times of initiation and termination. Times may be the global time (mission elapsed time), a time relative to when the specification is commanded, or a time relative to the start or completion of a designated process or graph.

## 4.5. Memory Management

A COSMOS dataflow process is not scheduled until sufficient memory for its execution and output tokens can be allocated. Static allocation of process execution memory and/or output token memory may be specified in the dataflow graph to avoid allocation overheads.

## 4.6. Non-dataflow Tasks

Dataflow and non-dataflow processes may coexist within a COSMOS system. The non-dataflow processes are either interrupt service routines or tasks managed by the multitasking kernel. Such a process is statically assigned to a single computer.

Dataflow and non-dataflow processes can exchange data and control information using the global (intercomputer) mailboxes and queues described in Appendix 1. If all of the processes are statically assigned to the same computer, local mailboxes and queues managed by the operating system kernel may be used.

## 4.7. I/O Operations

I/O operations may be performed by dataflow functions using the normal non-dataflow methods. Interrupt service routines and handlers may be installed in the operating system kernel. Processes may also manipulate I/O directly.

Functions may be restricted to execution on a single computer. Where fault tolerance is required, the I/O hardware and the dataflow function(s) are duplicated on two or more computers. The functions then vote on their operations by comparing input tokens. I/O may also be performed by non-dataflow processes.

## 4.8. Runtime Log

When a dataflow program is executed under COSMOS, a user-selected set of events may be logged. Each log entry is time-stamped. Loggable events include but are not limited to:

- an acquistion of a node by a computer
- a completion of a node by a computer
- a transfer of data between computers
- an occurence of a fault

This log is used for off-line analysis of the performance of specific applications and operating system overhead, as well as post-execution animation of the dataflow graphs.

## 4.9. Process Execution

The control of dataflow execution is distributed among the COSMOS computers. Each computer has a complete description of the dataflow graph.

Within each computer, dataflow processes are controlled by several COSMOS functions organized around a shared data structure describing the graph and its current state. Figure 1 shows the important data and control paths.

The dataflow control cycle begins when a graph node is fireable; follows with acquisition, execution, completion, and announcement of output tokens; and ends with the distributed updating of the graph state.

A graph node is fireable when all of its data and control inputs are available, its output arcs have room for its output tokens, and any timing constraints (such as starting time) have been met.

When a graph node is fireable, the Process Requester on each idle computer evaluates whether its computer has sufficient resources and rights (e.g., is a member of the node's computer group) to execute the corresponding process. Load balancing is implemented in the Process Requester.

Each Process Requester that decides that its computer could execute a fireable graph node broadcasts a "request for process acquisition" synchronization message specifying the graph node.

The Graph Manager on each computer receives all dataflow messages. When a process acquisition request is received, the request will be granted if the graph node is still fireable (it is possible that another computer is granted rights to the process before this process acquisition request is processed). All computers in the system have the same information and will reach the same conclusion about fireability. There is no need for an acknowledgment message.

If the Graph Manager grants a process acquisition request, it makes a note of the assignment in its copy of the Graph State and evaluates the fireability of the node's next generation. If the node is fireable again, it notifies the Process Requester that there still are fireable nodes.
If the Graph Manager grants a process acquisition request and if the request was broadcast from its own computer, it schedules a "Dataflow Template" process that executes the requested process. When the Dataflow Template process starts execution, it asks the Data Manager to collect the input tokens for the process.

The Data Manager controls the collection of copies of tokens for a computer, taking care of Data Transfer Network activities and of multiple requests for copies of the same token. If the data part of a token needs to be copied from another computer, the Data Manager sends a request to a computer containing the data. The Data Manager on that computer then returns a copy of the token's data. After the tokens are collected, the Dataflow Template process executes the code for the process.

When the code for a graph node has completed execution, the Dataflow Template process broadcasts a synchronization message announcing the process's completion and identifying the output tokens produced, if any. The actual token data is not broadcast, unless the

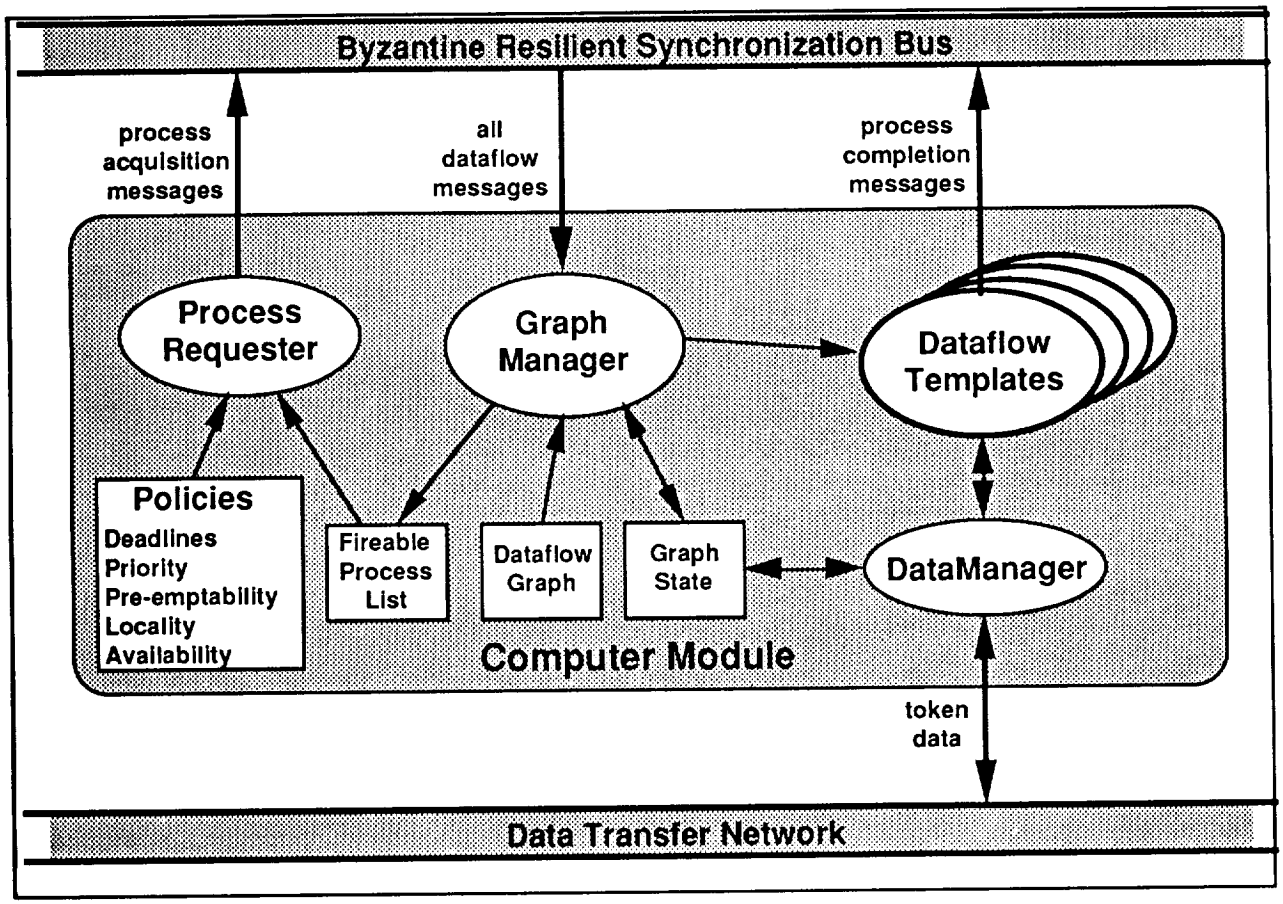


Figure 1. Dataflow Control Structure

token data fits into a single long word.[3] For critical token data, a mechanism does exist for storing redundant copies on another module.

When the Graph Manager receives a process completion message, it updates the process and token information in its copy of the Graph State. The fireability of graph nodes receiving tokens is reevaluated. If the Graph Manager finds that one or more graph nodes become fireable, it notifies the Process Requester that there are fireable processes to be evaluated.

This completes the summary of the dataflow control cycle.

[3] The single-word broadcast feature is used to implement the select control signal in a select group.

# 5. The COSMOS Programming Language

COSMOS uses a coarse-grained dataflow language wherein each graph node represents a process. A node's processing is commonly on the scale of an ordinary application "subprogram".

By using the dataflow paradigm at the process level, COSMOS gains several advantages:

- natural extraction of parallelism
- a solid framework for modular software development
- fault containment at process and computer boundaries
- ease of runtime code patching
- ease of checkpointing and rollback -- a COSMOS program is always in a defined state
- synchronization among tasks is defined exclusively by the data and control dependencies specified in the dataflow graph.

In the following sections, the COSMOS "dataflow" programming language will be detailed.

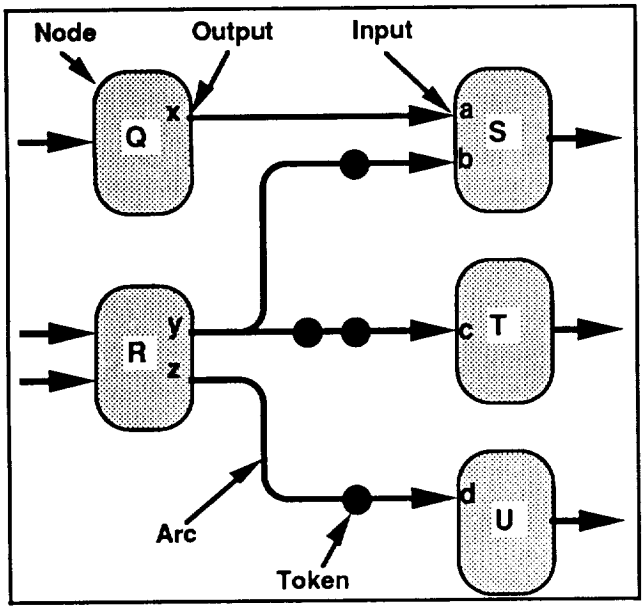


Figure 2. Basic dataflow graph.

## 5.1. COSMOS Dataflow Graphs

A dataflow graph consists of a collection of nodes, inputs, outputs, arcs, and tokens. The nodes represent the application processes and the inputs, outputs, and arcs define how such processes exchange data and control signals. Each node is connected to a set of inputs and to a set of outputs.

Arcs connect outputs to inputs and show *the only* flow of data among nodes. The presence of data packets is represented by tokens on arcs. Nodes, inputs, outputs, and arcs can each be assigned various attributes that affect the operation of a dataflow application. These attributes are specified in the dataflow graph.

Figure 2 shows a fragment of a dataflow graph. A node may have several inputs and outputs. If a single output is connected to several arcs, tokens generated by the output will be effectively duplicated on each output arc.

A dataflow node is ready for execution (fireable) as soon as all the required inputs are available, any timing constraints have been met, and there is sufficient memory for its execution and the output tokens.

The following sections describe the properties of tokens, arcs, nodes, inputs and outputs.

## 5.2. Tokens

Tokens are used to encapsulate data structures that are exchanged among graph nodes.

When an instance of a graph node is scheduled for execution, the input tokens that the node process receives are "reserved" for it. These tokens are not consumed until the process successfully completes. Thus, if a fault occurs, the input tokens are available for rescheduling of the node's execution.

All data tokens contain a header and a data body. The header describes the token attributes including its size, its checksum, a tag representing the data packet number, which arcs have consumed it, which computers have copies of its data, and a time stamp. The data body contains the actual data. Control tokens are represented only by a count, except for merged control inputs where properties of the control arc are involved and control tokens are queued as references to control arcs.

The data body can contain either computational data or executable code. The code for a graph node is supplied as an input token to the node. This is done primarily to simplify code patching, but also simplifies checkpointing,

code migration, and code preservation. The same mechanisms are used to handle both code and data tokens. When output data is announced as a result of a graph node's execution, each computer creates a token header to describe the data. Initially, only the computer that executed the node will have a copy of the actual data. The tokens on other computers will have null data bodies. Copies of the data bodies will be sent to other computers as needed for execution of processes receiving the tokens.

## 5.3. Arcs

Tokens logically move on arcs between graph nodes. Arcs are implemented as queues whose size may be user-specified. An arc's queue size defines the maximum number of tokens that may reside on the arc at any given time. If any of a node's output arcs are full, the node cannot be executed.

Initial tokens may be placed on arcs prior to the start of graph execution.

More than one arc may emanate from a single output of a dataflow node. When a token is generated on an output connected to more than one arc, the token is effectively replicated on each arc. The code implementing a graph node has no knowledge of the arcs connected to its outputs.

More than one arc may be connected to a single input of a dataflow node to form a compound input. The properties assigned to such an input determine how tokens are used from the arcs. This is explained in the sections below on Inputs.

An arc has user-specified attributes--consumability and update/uniqueness--which affect its operating characteristics.

If an arc is specified as consumable, tokens on the arc will be assigned once and only once to the node receiving the arc. The tokens will be assigned in order.

If an arc is specified as nonconsumable, the (logically) oldest token on the arc will be available for every firing of the node receiving the arc. A token on such an arc will be removed if the node that produced the token subsequently outputs an anti-token (see Section 5.11.1).

If an arc is specified as an update arc (not a unique arc), there will be at most one unreserved token on the arc. If such an arc already has an unreserved token on it when the node supplying the arc outputs another token, the new token will replace the older token. Thus, only the most recent token will be available on the arc.

If an arc is specified as a unique arc (not an update arc), tokens will remain on the arc until they are consumed by the node receiving the arc (or until the node supplying the arc outputs an anti-token).

These choices lead to four possibilities for both data and control arcs, as summarized in Figure 3.

A normal arc is both consumable and unique, and carries data tokens. The token data type must match the data types of the input/output to which the arc is connected.

### 5.3.1. [Non]Consumable Arcs

If the arc has the consumable parameter, the next input token on that arc is reserved when the next instantiation of the receiving node is assigned to a computer, and the token is consumed when the instantiation successfully completes. The next instantiation of the receiving node will use the next token on the arc.

The nonconsumable parameter is specified when tokens from an arc are to be used in multiple instantiations of the receiving node. When a token on such an arc is assigned to a node firing, it is immediately available for a subsequent firing of the same node. Nonconsumable tokens are used for constants, code tokens and sensor data that may be repeatedly used.

| | |
|---|---|
| Unique Consumable Arc | Normal first-in/first-out arc. Each token is used once and only once. |
| Update Consumable Arc | Only the most recent token is available. It will be consumed after it is used once. |
| Unique Non-consumable Arc | Not normally used. The oldest token is repeatedly used as input to the node receiving the arc. |
| Update Non-consumable Arc | Only the most recent token is available. It will not be consumed after it is used and will be available for subsequent firings of the node receiving the arc. Used for code and data constants. |

Figure 3. Dataflow Arc Attributes.

### 5.3.2. Unique/Update Arcs

The update attribute specifies that an arc can have at most one unreserved token. When a new token is announced on the output supplying such an arc, any unreserved token on the arc will be replaced by the new token. Tokens on the arc that are already reserved for a node firing are not affected until they are consumed (at which time they are replaced by the new token). An arc with both the update and the nonconsumable attributes is used for code tokens, where an update implies a code patch. Similarly, when an update arc is used for sensor data, new data replaces old data so that only the newest data is available.

If the unique attribute is used instead, all tokens must be used in order and cannot be updated or replaced. Tokens accumulate on an unique arc in their order of creation.

## 5.4. Nodes

COSMOS dataflow nodes describe coarse-grain functionality. A graph node has attributes such as execution priority and memory requirements. The code implementing a dataflow node is treated as an input token, as described below. A graph node also has inputs and outputs with attributes of their own, as described in subsequent sections.

### 5.4.1. Code for Graph Nodes

When a COSMOS system is initialized, the code for each node may be located on one, more than one, or all computers, at the user's discretion. The executable code that is used for a node firing is supplied as one of the node's inputs. This allows for simple and reliable replacement of code, i.e. "patching" while the program is executing.

Treating code as an input to a node permits process migration. If the code required by a node is not available locally, the code token will be requested, just as other data tokens are requested.

Code is normally supplied to a graph node from a nonconsumable update arc. The arc is nonconsumable so that when a code token is reserved for a node firing, it is immediately available for the next node firing. If it is required to replace the code for a graph node, a token containing the new code is announced on the output supplying the code arc. Subsequent firings will use the new code token. However, nodes already scheduled (or currently executing) will continue to use the old code.

Figure 4 shows the code token inputs for the nodes of a simple graph. In this case the three arcs supplying code tokens would in general come from different outputs of a node that distributes code tokens. Such a distribution

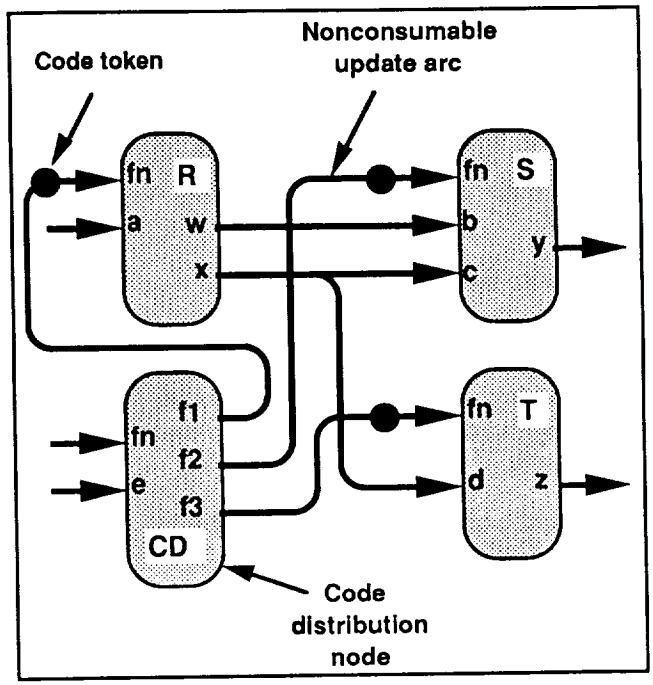


Figure 4. Code token inputs for graph nodes.

node might itself receive new code tokens from a spacecraft command system. If two or more graph nodes use the same code, their code token arcs come from the same output of a distribution node.

To simplify the figures, the code token input to nodes is not usually shown in COSMOS dataflow graphs.

### 5.4.2. Dataflow Node Granularity

In general, the dataflow paradigm may be used with a range of code granularities or average execution times for dataflow nodes. If the granularity is too fine, however, the overhead of managing the dataflow execution becomes significant compared to the average process execution time. On the other hand, if the granularity is too coarse, the benefits of the dataflow paradigm are reduced. An optimum granularity depends on a variety of factors including context switching overhead, fault-containment boundaries, size of local memory (or cache memory for code) and intercomputer communication delays.

In general, it is advantageous to select a uniform granularity for all dataflow nodes within an application since this simplifies load balancing.

## 5.5. Inputs

The code implementing a dataflow node receives its input data from tokens supplied by inputs attached to the node. This section describes the properties of such inputs.

The inputs to a dataflow node are arranged in a two-level hierarchy:

Arcs are bundled into "sub-groups";

Sub-groups are bundled into "groups."

Each input sub-group terminates one or more arcs. There are two types of input sub-groups: "Round-robin Merge" and "Voting."

Each input group consists of one or more input sub-groups. There are two types of input groups: "Priority Merge" and "Select."

The purposes of the input groups and sub-groups are listed in Figure 5. The input hierarchy is shown in Figure 6.

The priority merge input group supplies one token to each firing of the dataflow node. A select input group supplies two tokens to each firing of the dataflow node: a select value and a selected token. A process has no awareness of the input groups and sub-groups which feed it tokens: each input group is viewed as providing one (priority merge) or two (select) token values.

The code implementing a dataflow node has no knowledge of the node's input structure other than the number of tokens supplied for each firing. Thus, if more than one arc is connected to a priority merge input group the code will have no way of determining which arc supplied the group's token for a particular firing.

### 5.5.1. Firing Rule Summary

A dataflow node is ready for execution (fireable) as soon as all the required inputs are available, any timing constraints have been met, and there is sufficient memory for its execution and output tokens. The required inputs are available when all of the node's input groups are "ready." This section summarizes the rules determining when an input group is ready.

Priority Merge Group. Receives tokens from one or more input sub-groups. Such a group is ready if at least one of its sub-groups is ready. The sub-groups are given priorities so that if more than one sub-group is ready when the receiving dataflow node is fired, a token will be taken from the higher priority sub-group.

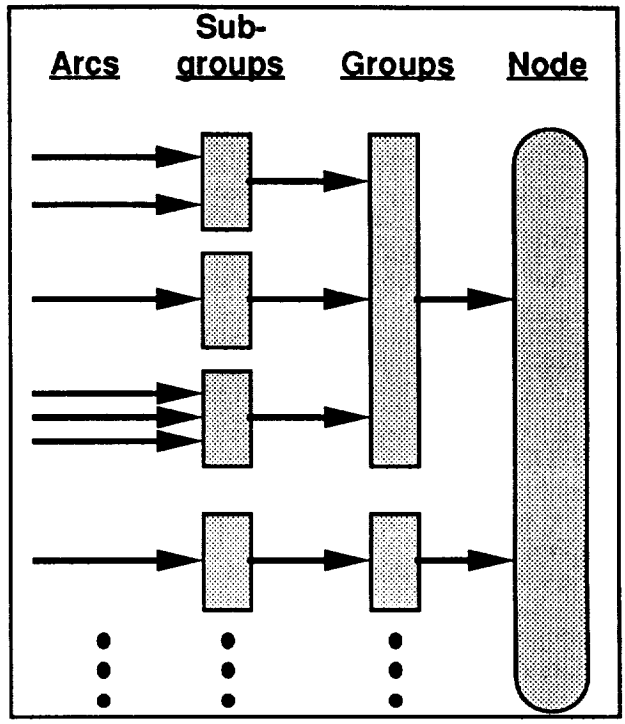


Figure 6. Input Hierarchy.

Select Group. Receives tokens from a selectable input sub-group. Contains a "select control" input sub-group and one or more "select body" input sub-groups. The arcs connected to the select control sub-group must contain

| | |
|---|---|
| Priority Merge: | Used to receive tokens from multiple sources. Sources are prioritized. |
| Round-robin Merge: | Used to receive tokens from multiple sources. Sources are of equal priority. |
| Select: | Used to receive tokens from multiple sources. A control token determines which source is used. |
| Voting: | Used to check for an exact match between two or more sources of tokens. |

Figure 5. Properties of node inputs.

tokens having integer values. The select body sub-groups are assigned indices: 0,1,2,... A select group is ready if its select control sub-group is ready and if the select body sub-group indexed by the select control token is also ready.

Round-robin Merge Sub-group. Receives tokens from one or more input arcs. Such a sub-group is ready if there is at least one token on at least one of the arcs. After a particular arc supplies a token for a node firing, the other arcs of the sub-group are checked for tokens that are available for the next node firing. Thus, the arcs are accessed with equal priority.

Vote Sub-group. Checks for an exact match between two or three input arcs. Such a sub-group is ready if at least two of the arcs have matching tokens.

Normal Input. Receives a token from a single arc. It can be considered a priority merge group fed by one round-robin sub-group connected to one arc. A normal input is ready when the arc supplying the input contains a token.

The following sections describe the input structures in more detail.

### 5.5.2. Data-Independent Inputs

In the simplest case, the fireability of a dataflow node does not depend on the values of its input tokens; it depends only on the availability of inputs and resources. Graphs or parts of graphs that use only these "data-independent" rules will have predictable behavior in terms of execution time, computer availability, and memory usage.

A graph node's firings will be data-independent only if all of its inputs are normal inputs, with or without voting.

#### 5.5.2.1. Normal Inputs

The basic input to a graph node is the "normal" input. Only one arc may be connected to a normal input. The input is ready when the arc contains an available token. When a node is executed, it will use one and only one token from each of its normal inputs.

Depending on the consumability attribute of the arc connected to the input, a token may or may not be reused by a subsequent firing of the node.

#### 5.5.2.2. Voting Sub-groups

COSMOS implements software "voting". Voting detects faults which occur during the execution of critical processes. Once detected, transient faults -- such as a bit-flip caused by a stray cosmic ray -- will be corrected by the operating system. Hard failures may require system reconfiguration.

In COSMOS, a user can specify which processes shall be voted -- only critical processes need incur any associated overhead. Note how COSMOS differs from other implementations where voting is obligatory. There must be exactly 3 processes supplying voted inputs (i.e. exactly 3 "voters"). A simple example is shown in Figure 7a. Figure 7b shows an example of triple modular redundancy.

Note that the code implementing the function of a node with voted inputs has no knowledge of the voting; COSMOS performs the proper voting and supplies the code with correct data.

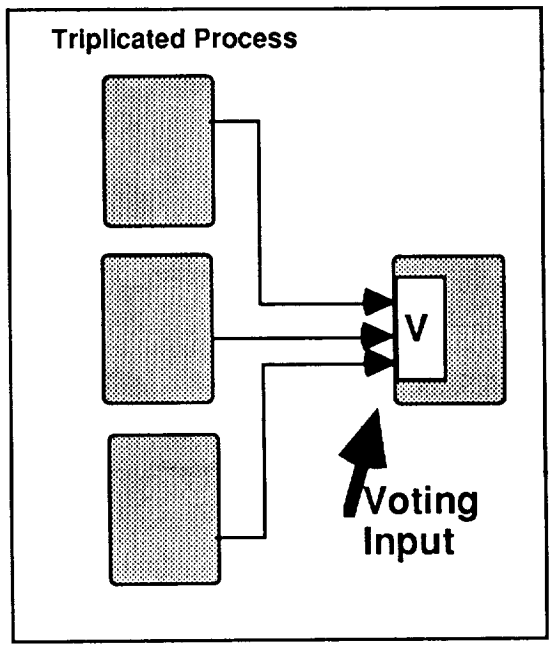


Figure 7a. Graph showing simplest use of voting.

In the event one voter disagrees with the other two, COSMOS records the mismatch along with an identification of the computer which produced it. All processors are notified of the mismatch. A user-defined fault-response process subsequently may deal with the faulty module.

In the event all voters disagree, COSMOS initiates a user-defined recovery process. This process may instruct COSMOS to roll back to a checkpointed state and resume execution.

Although not apparent at the graph level, in two special cases COSMOS actually performs a 2-way vote rather than a 3-way vote.

- If the first two voters agree, there is no need to wait for the third to complete in order to determine the vote outcome (if the third vote turns out to be a

mismatch, the mismatch will still be logged however).

- As an optional optimization, one of the voters may be set to "shadow" status; as a "shadow", its execution is not even initiated unless the first two voters both finish voting and disagree with each other.

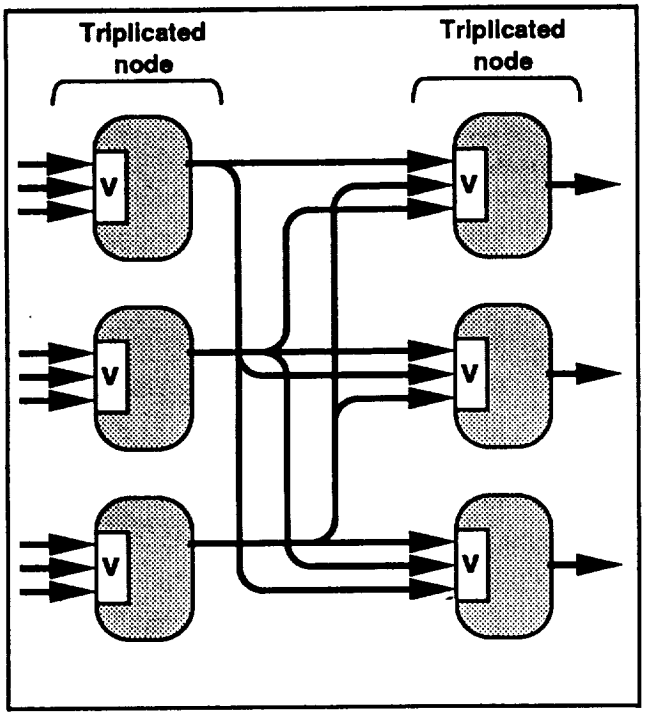


Figure 7b. Graph showing triple modular redundancy and voting.

### 5.5.3. Data-Dependent Inputs

It is also useful to have data-dependent firing rules. Here, a node may not require tokens on all input arcs, and sometimes (see "select groups") the value of a data token determines which input tokens are used.

Each input arc may have any of the attributes described in Section 5.3.

#### 5.5.3.1. Round-Robin Merge Sub-Groups

A Round Robin Merge sub-group is perhaps the most commonly used data-dependent firing rule. It is connected to two or more arcs of equal priority. After a token from one arc in a Robin Robin Merge is used, further tokens on that arc will not be used until one token from each of the other non-empty arcs is used. In other words, the first arc will be used next if (and only if) all other arcs in the Round Robin are empty.

#### 5.5.3.2. Priority Merge Groups

Priority Merge inputs (sub-groups) are given a numbered relative priority beginning at 0, for the highest priority. Tokens from arcs in higher priority sub-groups are consumed first. Tokens from arcs in lower priority sub-groups will not be used until all higher priority arcs are empty. Figure 8 shows an example of a Priority Merge input. In this case there is one high priority merge input (labeled with a "0") and one lower priority merge input (labeled with a "1").

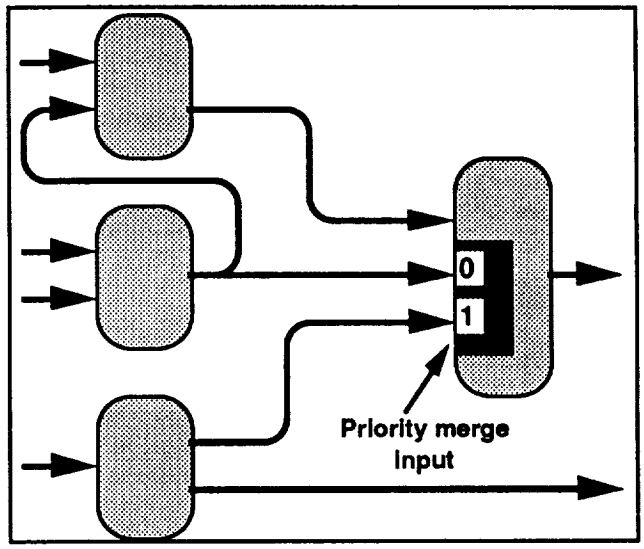


Figure 8. Priority Merge Input

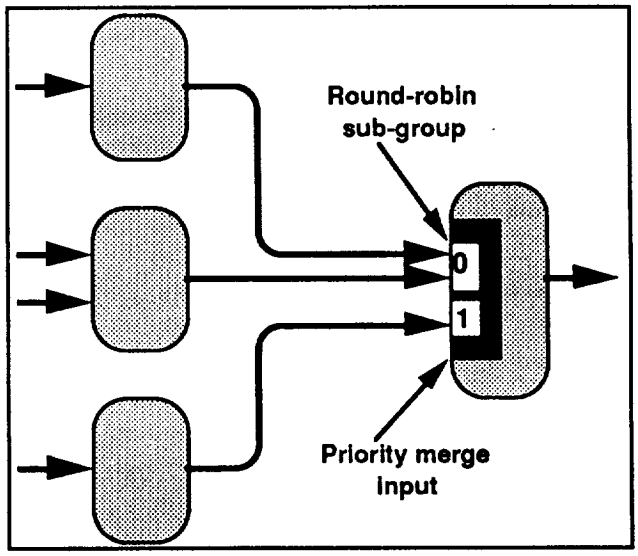


Figure 9. Round Robin Merge Input to a Priority Merge

Figure 9 shows another example of a Priority Merge Input. The lower priority merge input consists of a single normal input like in Figure 8, but in Figure 9 the high priority merge input consists of two arcs connected as a Round Robin Merge group. The combination of Priority and Round Robin Merges provides a powerful method of controlling token usage.

#### 5.5.3.3. Select Groups

A select input group contains a "select control" input sub-group and one or more "select body" input sub-groups.

The select body sub-groups are assigned indices: 0,1,2,...

The tokens received in the Select Control group must have integer values. This value selects the correspondingly numbered sub-group.

The code implementing the node will receive two inputs when it is fired, the Select Control Input token and a token from the selected sub-group.

Figure 10 shows an example of a select input. The token produced by Q on the select control arc S (of node R) must contain an integer value. Depending on this value, a token from one of the three sub-groups 0, 1, 2 will be selected. Note that the sub-group 1 is itself a round-robin merge input.

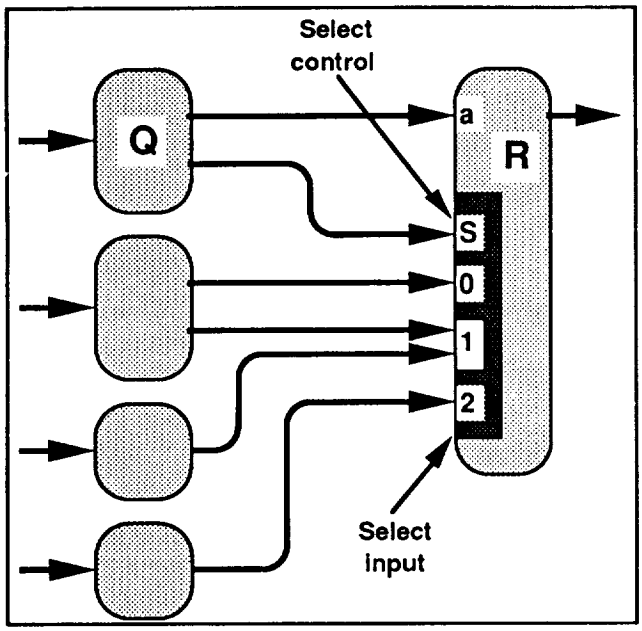


Figure 10. Select Input

## 5.6. Outputs

In general, a node firing may produce any number of tokens on an output. However, for predictable behavior, the number of tokens should be fixed for each output. This number is normally specified in the graph description of the output. The size of each output token is variable, and can be specified with the output description.

## 5.7. Simultaneous Multiple Node Instantiations

COSMOS allows multiple instantiations of a dataflow node to be simultaneously executed.

The maximum number of simultaneous instantiations is limited by the availability of input data, the queue sizes of the node's outputs, and the number of available computers.

It is not required that multiple instantiations complete execution in the order of firing. Each instantiation and the output tokens it produces are tagged to identify the instantiation's firing order. Effectively, tokens produced by a particular instantiation are not placed on their arcs until all previous instantiations have completed.

For predictable steady state performance, the operating point specification predetermines the number of simultaneous multiple instantiations needed for each node, the arc queue sizes, the initial control tokens required for each operating point, and the time between inputs for the graph containing the node. These constraints control the amount of parallelism that can be exploited for predictable performance.

## 5.8. Token Consumption/Generation

When a node fires, COSMOS "reserves" the input tokens for that instantiation and creates a "handle" that describes the instantiation and its inputs. When the node successfully completes execution and successfully broadcasts the availability of its outputs, the handle is replaced with the output tokens and the input tokens are consumed. In effect, the input tokens are replaced with the output tokens in a single transaction. Either the replacement occurs successfully or no change is made.

Thus, if the node execution fails to complete, the node can be re-executed on a different computer since the input tokens and a description of the instantiation are still available.

## 5.9. Token Preservation

When a node instantiation announces its output tokens, the actual data values of the tokens reside only on the

computer that executed the instantiation. This computer has the initial "preservation responsibility" for the data. Other computers request copies of the data values as required and may discard them when no longer needed. The computer with preservation responsibility for a token's data must not discard the data until all dataflow nodes using the data have successfully completed.

It is possible to specify that critical data be duplicated and preserved on a second computer as soon as the data is available. The output specification names a computer group that is responsible for the duplication.

## 5.10. Simulated Node Execution

A facility is provided that allows non-dataflow tasks to announce tokens into a graph as if a graph node had executed.

These simulated executions are normally used to introduce code and data tokens into a graph. This happens when the graph is first created and when code patches are made.

In a flight system, a command-processing task would perform the simulated executions when a command is received to patch or load a graph. In a development environment, simulated executions are announced by the Dataflow Debugger when loading tokens into a graph.

## 5.11. Special Operations

Normally COSMOS requires assignment of input tokens when a node fires, and announcement of all output tokens only when node execution completes.

However, COSMOS does support 3 special operations which permit the user to bypass this requirement. The 3 special operations allow graph nodes to receive inputs and to announce outputs without completing execution. An advantage to this approach is that arcs can be treated as "pipes" connecting processes. The price paid is the loss of some important fault tolerance features, including the ability to re-execute a failed node on a different computer. Therefore, any use of these special operations should be carefully considered.

### 5.11.1. Anti-tokens

An "anti-token" is a special type of data token used to flush previously announced tokens from all receiving arcs.

When an anti-token is announced on a node output, all tokens from previous node firings are deleted, except for tokens that have already been reserved for node firings. Tokens already generated by the early completion of subsequent node firings will not be affected.

### 5.11.2. Partial Completion

The code for a graph node may request that one or more sets of tokens be announced before node execution completes. Each such announcement is a partial completion.

If an anti-token is announced on an output, all previous tokens are deleted, including any produced by previous partial completion of the current instantiation.

### 5.11.3. Multiple Inputs from an Arc

The code for a graph node may request assignment of further tokens to specific inputs via a "New Input" facility.

If a new token is available, the token previously assigned to that input is consumed and the new token is assigned in its place for the node firing.

If a new token is not available, the token previously assigned to that input is not affected and the code receives an appropriate status flag and continues execution.

# 6. Performance Optimization

COSMOS supports several performance optimizations for the dataflow graphs described in the previous section. Dataflow processes may be assigned priorities. One of a triad of voting processes may be labeled a "shadow" process which fires only if the two primary processes disagree. In order to regularize execution patterns and impose timing constraints, a control graph may be used (by superimposing it over the dataflow graph).

A set of optimizations will be relevant only to a specific hardware configuration and operating environment. These optimization sets are termed "operating points"; the active operating point will change as either the hardware configuration changes--as one computer fails or another is brought on-line, for example--or the operating environment changes.

## 6.1. Control Graphs

COSMOS control graphs serve to regulate interprocess timing relationships between the nodes of a COSMOS dataflow graph. Like COSMOS dataflow graphs, COSMOS control graphs consist of process nodes, outputs, arcs, and inputs. Control tokens are passed from control output to control arc (control edge) to control input.

### 6.1.1 Properties of Control Arcs

Control arcs have three principal properties: generational offset, delay time, and arc enable. Generational offsets specify that a control token of generation n will be used to enable a generation n + generational offset target process.

Delay time specifies a minimum interval between the enabling of an instantiation of a process and the announcement of tokens by that instantiation. If a process has multiple DELAY arcs feeding it, then these arcs are treated as a merge. The ENABLE attribute specifies which control output will receive tokens as the result of a process firing. The ENABLE attribute is used in conjunction with the DELAY attribute to implement scheduling decisions.

### 6.1.2 Control Arc Merges

Each process may have a single merge input, in addition to its normal control inputs. The merge is a temporal merge: control tokens are used in the order that they are received. All input arcs with the DELAY attribute are merged.

## 6.2. Operating Points

In data-independent graphs which can be performance-optimized with the ATAMM rules, maximum performance can be assured by dynamically altering the graph, primarily by placement of control arcs. The operating point parameters can be predetermined by analysis of the COSMOS dataflow graphs to be executed and then stored in a table of operating points for COSMOS to dynamically apply.

The operating point for a single graph or for a set of graphs determines values for TBI, TBIO, TBO, and R.

- TBI is the time between inputs to a graph.
- TBIO is the time between the input of a data packet to the graph and the generation of the corresponding output result from the graph.
- TBO is the steady state time between successive data outputs from the graph.(1/TBO represents steady state throughput).
- R is the number of available computers which can work on the graph.

Given a number of available computers, the desired combination of TBO and TBIO values is ensured by the use of control arcs, arc queue sizes, initial tokens, and the number of simultaneous node instantiations.

A desired combination of TBO and TBIO values is ensured by controlling the time between inputs (TBI) accepted by the first node in the graph. This equals TBO in steady state. Control arcs, arc queue sizes, initial tokens, and the number of node instantiations allow user-specified tradeoffs between TBIO and TBO over a range of available computers.

For multiple graphs being executed simultaneously, each operating point consists of a value of TBIO and TBO for each graph and one value of R. There can be a number of operating points for each value of R.

The choice of operating points can be used to favor certain graphs in specific situations. A predetermined strategy may be developed to favor certain graphs as computers fail (graceful degradation) and/or to provide predictable user-selected performance for specific graphs for a dynamically changing number of computers.

COSMOS dynamically shifts to a different operating point whenever the number of available computers changes or mission requirements change. When such an event happens, an operating point change is made and the user's application is notified, and a further operating point change may result. A shift in the operating point will generally involve the insertion and deletion of control arcs, the number of multiple instantiations of a node, the queue sizes on arcs, and the injection interval TBI. Additionally, a miscellany of other characteristics are subject to change.

## 6.3. Miscellaneous Mutable Parameters

As a result of an operating point change, a variety of optimization parameters may change in value. At the process level, the range of modules (module groups) on which a process is allowed to execute may change; the maximum number of concurrently executing instantiations of a process may change; and a voted process may become a shadow process. Process priorities may change. The length of time that replicated token data is kept on a module for a given output may change, as may the queue size for a given output.

# 7. The COSMOS Programming Environment

A suite of tools are included in COSMOS to assist the programmer in design, testing, debugging, and performance enhancement. Figure 11 shows an overview of the COSMOS tool set. Figure 13, at the end of this section shows the exchange of data files among these tools.

For developing code for a particular hardware target, COSMOS would be used in conjunction with a commercial high-level language (e.g. C) development environment. This environment must include a text editor, as well as a compiler for the high-level language that is used.

The remainder of this section describes the COSMOS tool set in more detail.

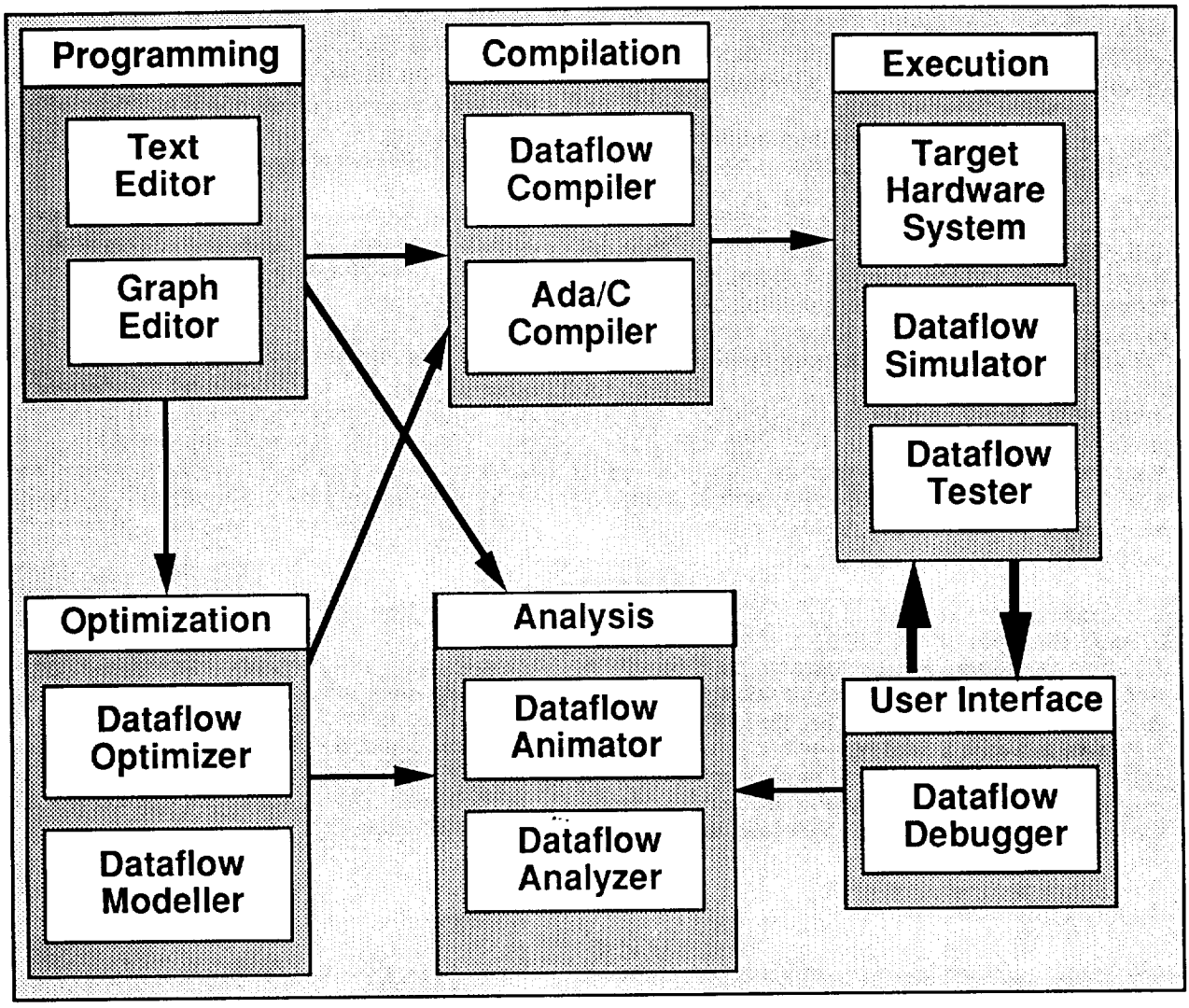


Figure 11. COSMOS Tools

## 7.1. Programming

The programming environment consists of a commercial text editor and a dataflow graph editor.

### 7.1.1 The Text Editor

Any existing or commercial text editor which can output an ASCII text file may be chosen by the programmer -- "vi" or "emacs" will suffice, for example.

The text editor is used to enter and modify a high level language function for each graph node. For example, the software implementing a graph node currently can be written in an extended version of C. Support for C++ and Ada is also planned, as the particular choice for a high-level language in a specific COSMOS implementation will depend on the availability of compilers and associated development tools for specific target hardware.

The extensions to C (which have been implemented as C macros) add dataflow-related keywords for referring to input and output tokens, initialization behavior, and the like. A dataflow function begins with definitions of the data structures for input and output tokens. Then the names and types of the inputs and outputs are declared, followed by the body of the function. Refer to the example of Figure 12. The function cannot have any "side effects" such as updating a global variable.

### 7.1.2 The Graph Editor

Dataflow graphs are created with the COSMOS Graph Editor. This graphical tool allows graphs to be drawn and allows the attributes of nodes, inputs, outputs, and arcs to be specified.

The Graph Editor can output 2 types of ASCII text files suitable for input to the Dataflow Compiler and the Dataflow Optimizer; one file describes the connectivity and attributes of the nodes, while the other file describes the initial token values. In addition, the Graph Editor can both save and open a third type of file, a "working" file which fully describes the graph picture; unlike the files output for the Dataflow Compiler, it is possible to restore on screen a previously saved graph from the "working" file.

## 7.2. Compilation

The compilation tools generate code for the target system, the Dataflow Tester, and the COSMOS Simulator. The principal tools are the Dataflow Source Code Filter, the Dataflow Compiler, and a compiler for the application's language.

### 7.2.1. The Dataflow Source Code Filter [4]

The Dataflow Source Code Filter pre-processes certain COSMOS language extensions (see the description of the Text Editor and Figure 12) into the application's native language. The resulting source code is compiled[5] and linked into an object module. The object module must be relocatable, either as a fully relocatable object module or as a loadable object module which can be bound to an arbitrary address.

For the MAX hardware target, it was necessary to compile the code into assembler format, convert that to relocatable form, assemble and link the result, and finally convert the linked module to a downloadable ASCII format. Not all processors support relocatable object code, however. For some target processors, it may be necessary to incorporate a loader into COSMOS to convert code tokens from a relocatable format to one which is linked to absolute addresses. The details will vary depending on the language and compilers used, as well as the processor architecture.

The Dataflow Source Code Filter also generates a file describing the dataflow inputs and outputs which is used by the Dataflow Compiler to ensure consistency with the graph description.

### 7.2.2. The Dataflow Compiler

After a dataflow graph has been created and its code has been filtered and compiled, the Dataflow Compiler combines them into a form ready for downloading to the target COSMOS platform. The reader may find it easier to think of the Dataflow Compiler as a linker rather than as a compiler.

The Dataflow Compiler takes as inputs a graph description file, the object modules for code tokens, the token description files, the optimization files, and subsidiary files used for consistency checking. The Dataflow Compiler checks for consistency of the code and the graph. Input and output data types specified in the graph are verified as being identical to the types specified for the object modules, and the number and size of output tokens are also checked.

---

[4] For simplicity's sake, the Dataflow Source Code Filter, and the files it produces, are not shown in Figure 11 or Figure 13.

[5] The word "compiled" here means compiled by a commercial target hardware compiler -- not compiled by the Dataflow Compiler.

The primary output of the Dataflow Compiler is a file suitable for download to the target hardware system. This output may include a description of the entire graph, a complete set of operating points for the graph, code and other data tokens, or a combination of these elements.

```
/* IncSq_1.t              A simple dataflow function.*/
#include "CodeTokens.h"    /* Include dataflow definitions.*/
/*-------------------------------------------------------*/
/* Declare the data structure of the tokens: */
typedef struct                    /* Generally imported by a ".h" file */
   {
   uslong  value, square;
   }
SQUARES;
/*-------------------------------------------------------*/
INPUT   SQUARES a;          /* Declare the dataflow input. */
OUTPUT  SQUARES x;          /* Declare the dataflow output. */
/*-------------------------------------------------------*/
CODE_TOKEN  (IncSq_1)       /* Declare the dataflow function.*/
   {
   /* Declare any local variables here */
   START_CODE;              /* Marks end of local declarations.*/
   x = NEW_TOKEN ();        /* Create the output token.*/
   x.value  = a.value + 1;
   SetSquare ();
   END_FIRING;              /* End node firing and
                               announce output token. */
   }
/*-------------------------------------------------------*/
/* Local function to set part of the output token's value.*/
static void SetSquare ()
   {
   x.square = x.value * x.value;
   }
```

Figure 12. Example Dataflow Source Code.

## 7.3. Execution & Simulation

The primary execution tool is the COSMOS operating system itself. In addition, 2 simulators are available that execute the actual application code. The Dataflow Tester provides functional testing and verification of dataflow applications. The Dataflow Simulator provides a detailed simulation of all aspects of COSMOS operation, including interprocessor communication.

### 7.3.1. The Dataflow Tester

An important feature of COSMOS is that the code for a graph node may be tested in isolation. Since side effects are not allowed, the functional properties of a node are completely specified by the relationship between its inputs and its outputs.

The Dataflow Tester supports the separate testing of code. The programmer provides test vectors for a node in the form of a series of input tokens. The Dataflow Tester then executes the node in a simulated dataflow environment and records the generated outputs. A verification file is produced that describes the inputs and the resulting outputs. The Dataflow Tester operates in any C or Ada environment independent of the target hardware.

The inputs to the Dataflow Tester are the dataflow graph and the dataflow code for the nodes. When testing individual nodes, it is not necessary that the code for all nodes be present, i.e. it can be used for incremental development. The Dataflow Tester also performs all of the consistency checks that the Dataflow Compiler performs.

The Dataflow Tester may also be used to test the free running execution of a whole graph or any part of one. Since it implements the dataflow rules and is provided the graph description, it can pass tokens produced by a node to those nodes that use them as input.

### 7.3.2. The Dataflow Simulator

The Dataflow Simulator provides a complete functional simulation of an application running on COSMOS in a multicomputer system. The simulator can be easily ported to any 'C' environment.

The simulator includes the complete COSMOS operating system code, excepting that which deals with relocatable code tokens and that which is hardware-specific. Input to the simulator is the compiled source code for graph nodes and the graph description for the application to be simulated.

The Dataflow Simulator will simulate execution on any desired number of computers to provide a detailed simulation of an application's execution behavior.

The hardware environment is fully simulated:

the given number of computers;

the topology of the data transfer network;

the synchronization network;

serial I/O ports.

The underlying multitasking executive on each computer is also simulated, as is the passage of time. Simulated execution times can be attached to most system events, so that the simulator can be "tuned" to match different hardware environments and so that simulation behavior is faithful to the behavior of the targeted hardware. The macros which support simulation of time can be replaced by ones which measure time for a COSMOS kernel running on a target system.

Ideally, the Dataflow Simulator or the Dataflow Tester could be used to generate estimates of node execution times which could be used as inputs for the design tools. Unfortunately, microcomputer and workstation platforms rarely support the fine-grained timing required, and the actual target hardware often differs from the development platform in terms of instruction sets and other characteristics. Node execution times can be refined when the code is actually run on the target hardware.

## 7.4. Debugging

The Dataflow Animator and the Dataflow Debugger give visibility into the behavior of a program (but not performance -- see section 7.5 for that). A programmer will most likely use both the Dataflow Debugger and the Dataflow Animator to debug a COSMOS program.

The Dataflow Debugger provides a user interface for COSMOS. Through it, the running behavior of a COSMOS system can be observed, faults injected, and a variety of debugging parameters may be set or cleared. The Dataflow Animator is useful for detecting bugs at the graph level.

### 7.4.1. The Dataflow Animator

The Dataflow Animator displays the graph as entered into the Dataflow Editor and shows the flow of tokens among the graph nodes, the computers executing each node firing,

and the data traffic. Graph execution may be replayed forwards or backwards in time.

The Dataflow Animator operates on log files that describe dataflow events which occur during an application's execution. The COSMOS runtime system, as well as the Dataflow Simulator, Dataflow Tester, and Dataflow Modeller, can generate these log files.

### 7.4.2. The Dataflow Debugger

The Dataflow Debugger provides interactive debugging with COSMOS, the Dataflow Tester and the COSMOS Simulator. It is often used in conjunction with an existing commercial debugger.

The Dataflow Debugger can examine and control the execution of a dataflow application. It has features such as single-stepping through a graph's firings, examining its tokens, reporting the current graph state, and examining all operating system tables and states.

The Debugger works with the COSMOS operating system, the Dataflow Tester and the Dataflow Simulator.

When a dataflow program is executed on real hardware or in the Dataflow Simulator, a runtime log may be produced if desired. This information can include which nodes executed on which computers and what data was transferred between computers. While usually used as input to the Analyzer or Animator, the Debugger can print such logs in a readable form.

In a ground-based development environment, the full debugger is installed in the COSMOS system. In a flight application a streamlined version would allow a spacecraft telemetry system to interrogate and control dataflow execution.

## 7.5 Performance Analysis And Optimization

The Dataflow Optimizer, Analyzer, and Modeller all work together to provide performance analysis and optimization. The goal is to determine graph operating points and resource requirements.

### 7.5.1 The Dataflow Optimizer

The Dataflow Optimizer automates the analysis of data-independent graphs and determines the necessary resource dependent graph modifications necessary to optimize performance. These graph modifications serve to optimize resource usage by an application.

The initial graph input is generally provided from the Graph Editor as a description of the nodes, node execution times, and data arcs.

The tool models dataflow graphs so that optimum performance and sufficient resource requirements (number of computers) can be predicted. Information gathered from this tool can guide the decomposition of algorithms and determine the operating strategies required for different runtime condition levels, e.g. computer availability and input data rate.

The design process enables the user to provide control arcs and control tokens for all possible hardware configurations, thereby allowing automatic selection of operating points in response to a change in hardware configuration for real-time response; the application programmer is also given the tools to change operating points for fine-tuning of a running application.

The Optimizer's output is a description of the performance bounds for the group of graphs to be executed. Also, an operating point table is produced to enable dynamic performance optimization in the event the number of available computers changes. The operating point table is incorporated into an optimization file suitable for input to the Dataflow Compiler.

The necessary phasing of graph nodes and graph inputs needed to attain the desired performance can be determined. This phasing is generally different from that required only for the flow of actual data.

### 7.5.2. The Dataflow Analyzer

This tool analyzes program performance. It displays timelines showing each computer's activity, each node's activity, resource utilization and system overhead. [11] It also calculates computational performance metrics, including the average throughput in million instructions per second (MIPS).

The Dataflow Analyzer operates on log files that describe dataflow events which occur during an application's execution. The COSMOS runtime system as well as the simulators (Dataflow Simulator and Dataflow Tester) can generate these log files.

### 7.5.3 The Dataflow Modeller

The Dataflow Modeller provides architecture-independent modeling of dataflow graph behavior, serving to quickly verify the performance predictions and graph optimizations proposed by the Dataflow Optimizer. In addition, typical graph-operation parameters such as delays imposed by shared-resource contentions can be modeled. Accounting for these added delays, which are not modeled by the

Dataflow Optimizer, ensures that added overhead can be accommodated for a given number of processors.

The Dataflow Modeller output is a log file of graph-operation events to be retraced and evaluated by the Dataflow Analyzer or the Dataflow Animator.

## 7.6. Tool File Exchange

Figure 13 shows the primary files exchanged among the COSMOS tools.

Code files are ASCII text files generated by an editor of the user's choice. These files contain the source for each graph node.

Graphs and tokens are described in text files[6] generated by the Graph Editor. Optimization files describe the operating points, control arcs, and associated graph parameters.

Log files record the execution of an application. They include the timing of all dataflow events, the identity of the computer executing each node firing, and can record network traffic in systems with data transfer networks and/or synchronization buses.

[6] A separate document, entitled *The COSMOS Text Graph Language*, describes the file format in detail. As of this writing it is still in progress.

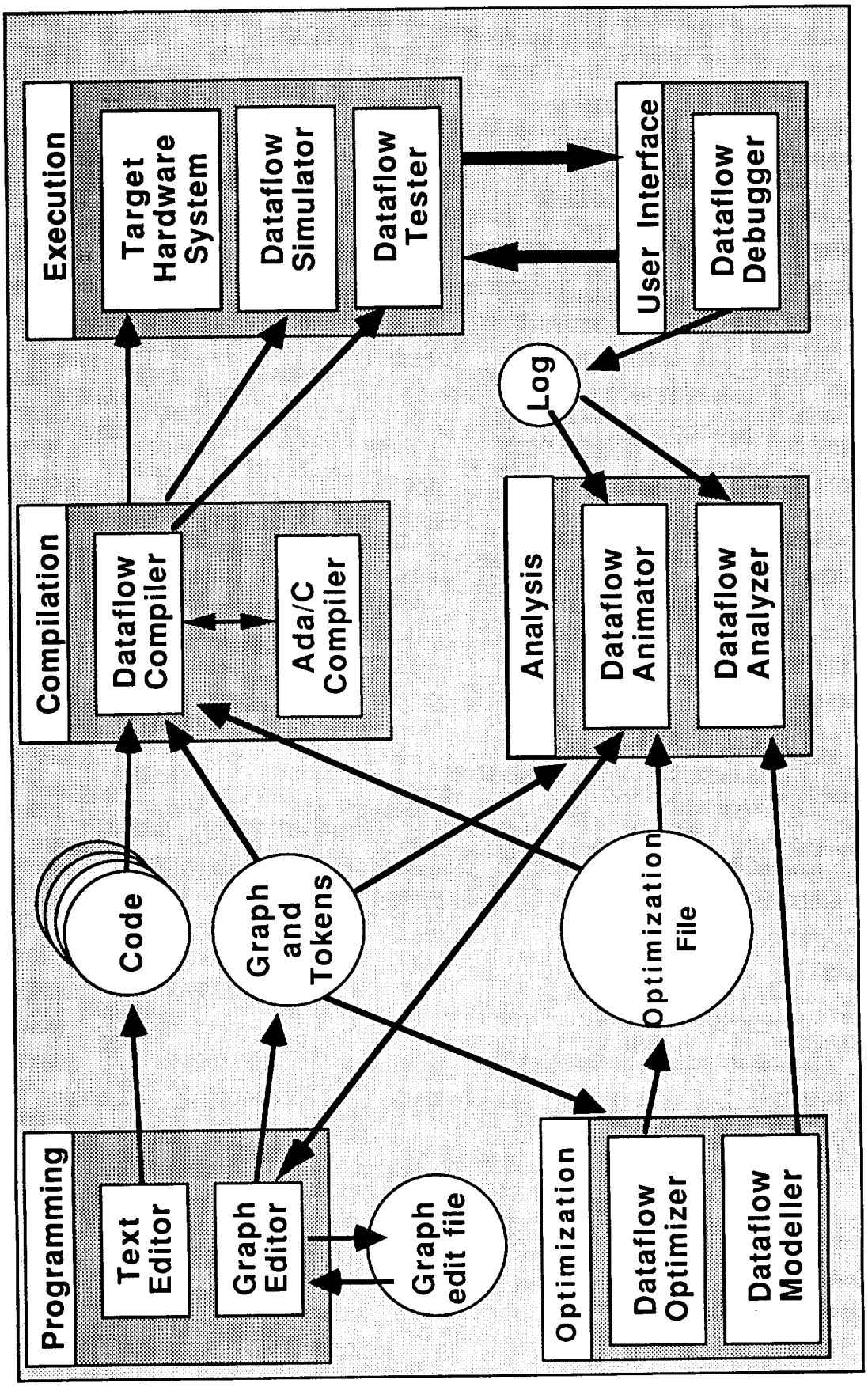


Figure 13. COSMOS Tool File Exchange.

# 8. References

[1] R. R. Mielke, John W. Stoughton, and Sukhamoy Som, *Modeling and Performance Bounds For Concurrent Processing*, NASA Contractor Report 4167, Grant NAG1-683, 1988.

[2] P.J. Hayes, R.L. Jones, H.F. Benz, A.M. Andrews, J.W. Stoughton, R.R. Mielke, M. Malekpour, and P.R. Appleget, *VHSIC Multiprocessor Implementation of the ATAMM Strategy*, GOMAC91/1991 Digest of Papers, 521, 1991.

[3] R. Mielke, J. Stoughton, S. Som, R. Obando, M. Malekpour, and B. Mandala, *Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification*, NASA Contractor Report 4339, Cooperative Agreement NCC1-136, 1990.

[4] S. Som, R. Mielke, R. Obando, J. Stoughton, P.J. Hayes, R.L. Jones, *Throughput Enhancement by Multiple Concurrent Instantiations in the ATAMM Data Flow Architecture*, Proceedings of the ISMM International Symposium on Computer Applications in Design, Simulation, and Analysis, 71, Las Vegas, NV, 1991.

[5] R.L. Jones, P.J. Hayes, A.M. Andrews, S. Som, J. W. Stoughton, and R.R. Mielke, *Enhanced ATAMM for Increased Throughput Performance of Multicomputer Data Flow Architectures*, Proceedings of the IEEE NAECON 91, Vol. 1, 238, Dayton, OH, 1991.

[6] P.J. Hayes, R.L. Jones, H.F. Benz, A.M. Andrews, and M.R. Malekpour, *Enhanced ATAMM Implementation on a GVSC Multiprocessor*, GOMAC92/ 1992 Digest of Papers, 181, 1992.

[7] D. Blough, L. Alkalaj, and B.F. Lewis, *Clock Synchronization in the Common Spaceborne Multicomputer Operating System*, Technical Report ECE-93-05, Department of Electrical and Computer Engineering, University of California (Irvine), 1993.

[8] B. Chor and B.A. Coan, *A Simple and Efficient Randomized Byzantine Agreement Algorithm*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 6, 1985.

[9] R.D. Rasmussen, G.S. Bolotin, N.J. Dimopoulos, B.F. Lewis, R.M. Manning, *Advanced General Purpose Multicomputer for Space Applications*, Proceedings 1987 Conference on Parallel Processing, 54, 1987.

[10] B.F. Lewis and R.L. Bunker, *MAX: An Advanced Parallel Computer for Space Applications*, Second International Symposium on Space Information Systems, 769, 1991.

[11] R.L. Jones, J.W. Stoughton, and R.R. Mielke, *ATAMM Analysis Tool*, NASA Contractor Report 187625, 1991.

# Appendix 1. Intercomputer Services

The operating system kernel provides the basic multitasking services available to a single computer. Built upon the kernel are intercomputer services to allow processes and interrupt service routines on one computer to communicate with processes and interrupt service routines on other computers. However, allowing a process thread to cross computer boundaries is not supported in COSMOS.

The basic intercomputer services are mailboxes, queues, semaphores, and event flags.

## 1.1. Mailboxes and Queues

Intercomputer mailboxes and queues are objects that can hold data records (or messages). Queue lengths may be predefined. A mailbox is a degenerate queue holding at most one item.

Software on any computer can post data to an intercomputer queue by allocating a local memory area for the data, supplying the data, and then posting the data to the desired queue. Once the local data area has been posted to a queue, it can no longer be accessed by the software that posted it. The actual queue will contain a handle identifying the posting computer and the local memory area within that computer.

When another (or the same) piece of software pends on the queue, COSMOS will allocate local memory on the pender's computer, copy the data there, and delete it from the poster's computer after successful copying. The pending software will receive a pointer to the copy of the data on its computer. If the pender and the poster are on the same computer, no copying is required and the pender receives a pointer to the original data.

An intercomputer mailbox or queue is generally resident on a single computer. However, redundant copies can be maintained on different computers for increased fault tolerance. The mailbox or queue could be identified by a handle, without regard to its residency.

## 1.2. Event Flags

Intercomputer event flags are objects distributed by synchronous messages. If software on one computer broadcasts a message modifying an event flag, all computers will record the change. Any number of processes can pend on an event flag or a logical combination of event flags.

# Appendix 2. Definitions and Acronyms

| | |
|---|---|
| ASCII | - American Standard Code for Information Interchange |
| ATAMM | - Algorithm to Architecture Mapping Model |
| AMOS | - ATAMM Multicomputer Operating System |
| COSMOS | - Common Spaceborne Multicomputer Operating System |
| computer | - a processor in the multicomputer system; a central processing unit with local memory; "computer" and "processor" are used interchangeably in this document |
| control arc | - an arc which has no data and is typically used for control purposes only; may be associated with timing information. |
| enabled | - status of a node when all input arcs and output arcs meet the requirements for firing and execution |
| execution | - carrying out the function of a graph node; this includes the reading of data, consumption of input tokens, functional computations of the node code, writing of the output data, and depositing of output tokens |
| fire | - the beginning of execution of a graph node when the node is bound to a computer and the input data tokens become reserved |
| handle | - a logical name |
| HYPHOS | - a dataflow operating system developed by JPL (not an acronym) |
| JPL | - Jet Propulsion Laboratory |
| LaRC | - Langley Research Center |
| MAX | - a multicomputer architecture developed by JPL; contains both a control bus and a data bus for intercomputer communication (not an acronym) |
| node | - a functional block of code in a data flow graph |
| pend | - accessing data such as that existing in a queue |
| post | - depositing data such as that into a queue |
| process | - a function to be executed; represented by a graph node |
| processor | - a computer; see "computer" |
| reserved token | - a token representing a data packet which has been assigned to a computer for graph node execution |
| task | - a function defined formally in the Ada language |
| VHSIC | - Very High Speed Integrated Circuit |

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | February 1994 | Technical Memorandum |

**4. TITLE AND SUBTITLE**
Common Spaceborne Multicomputer Operating System and Development Environment

**5. FUNDING NUMBERS**
233-01-03

**6. AUTHOR(S)**
L. G. Craymer and B. F. Lewis (JPL)
P. J. Hayes and R. L. Jones (NASA LaRC)

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Jet Propulsion Laboratory
Pasadena, CA and
NASA Langley Research Center
Hampton, VA 23681-0001

**8. PERFORMING ORGANIZATION REPORT NUMBER**
JPL Report #D-11525

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
National Aeronautics and Space Administration
Washington, DC 20546-0001

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
NASA TM-109092

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Subject Category 61
Unclassified - Unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

A preliminary technical specification for a multicomputer operating system is developed. The operating system is targeted for spaceborne flight missions and provides a broad range of real-time functionality, dynamic remote code-patching capability, and system fault tolerance and long-term survivability features. Dataflow concepts are used for representing application algorithms. Functional features are included to ensure real-time predictability for a class of algorithms which require data-driven execution on an iterative steady state basis. The development environment supports the development of algorithm code, design of control parameters, performance analysis, simulation of real-time dataflow applications, and compiling and downloading of the resulting application.

**14. SUBJECT TERMS** Multiprocessing, dataflow, parallel processing, operating system, real-time computing

**15. NUMBER OF PAGES**
33

**16. PRICE CODE**
A03

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |